Numerical Optimization with PETSc/TAO: The Multidimensional Rosenbrock Problem

A practical introduction to large-scale gradient-based optimization

At a Glance

Questions	Objectives	Key Points
1. What is optimization?	Understand the basic principles	Optimization seeks the inputs of a function that minimize it
2. Why use gradient-based methods?	Learn about trade-offs in algorithm choice	Gradient-based methods find local minima with the fewest number of function evaluations
3. How can we compute gradients?	Evaluate different sensitivity analysis methods	Applications should provide analytical gradients whenever they can
4. How to solve constrained problems?	Introduce constraints to the optimization problem	Constraints can change the solution or introduce new local minima

Introduction to Optimization

Optimization algorithms seek to find theinput variables or parameters (referred to as "control", "design" or "optimization" variables) that minimize (or maximize) a function of interest.

$$\underset{p \in X}{\text{minimize}}\ f(p)$$

- p: optimization variables
- $f : X \mapsto \mathbb{R}$: objective function

Optimization is a *huge* topic

Sub-disciplines branch off based on the space X —

- Discontinuous: combinatorial optimization, integer programming
- **Continuous**: (today)
  - **Unconstrained:** $X = \mathbb{R}^n$
  - **Constrained:** $X = \{x \in \mathbb{R}^n : g(x) = 0, h_1(x) \geq 0, h_2(x) \geq 0, \dots\}$

— and the the objective function f —

- Convexity: convex optimization
- Stochasticity
- **Smoothness:** (today)
  - Continuous: zeroth order methods
  - **Differentiable:** first order methods

Numerical Optimization by Nocedal & Wright is a standard reference for the types of methods discussed today.

In this lesson, we focus on gradient-based optimization methods -- methods that utilize information about the sensitivity of the objective function with respect to its inputs.

Solutions to this problem are found where the gradient of the objective function is zero,

$$\nabla_p f(p) = 0.$$

However, this is only a *necessary* but not sufficient condition for optimality given that other stationary points (e.g., maxima) also satisfy this condition.

Sequential Quadratic Programming (SQP)

To find local minima for the above problems, we replace the original problem with a sequence of quadratic subproblems,

$$\underset{d}{\text{minimize}} \quad f_k + d^T g_k + \frac{1}{2} d^T H_k d^T,$$

where $g_k = \nabla_p f(p_k)$ is the gradient, $H_k = \nabla_p^2 f(p_k)$ is the Hessian, $d \in \mathbb{R}^n$ is the search direction, and the k subscript denotes evaluation at the iterate p_k.

The exact solution to this quadratic subproblem is the inversion of the Hessian onto the negative gradient,

$$d = -H_k^{-1} g_k .$$

In order to avoid non-minimum stationary points, we also seek to find a step length α that approximately minimizes the objective function along the line defined by the search direction,

$$\underset{\alpha}{\text{minimize}} \quad \Phi(\alpha) = f(p_k + \alpha d).$$

This scalar minimization problem is called a "line search", and is categorized as a "globalization" method because it helps maintain consistency between the local quadratic model and the global nonlinear function.

The SQP class of algorithms can be summarized with the pseudocode:

for k=0,1,2,... **do**
  $\min_d \quad f_k + d^T g_k + 0.5 d^T H_k d$
  $\min_\alpha \quad \Phi(\alpha) = f(p_k + \alpha d)$
  $p_{k+1} \leftarrow p_k + \alpha d$
end for

In this approach, different approximations to the search direction solution yield different members of the SQP family:

- **[Truncated] Newton:**

  $$d = -H_k^{-1} g_k$$

  with the Hessian inverted [iteratively (e.g., Krylov methods) using dynamic tolerances].
- **Quasi-Newton:**

  $$d = -B_k g_k, \quad B_k \approx H_k^{-1}$$

  with low-rank updates based on the secant condition.
- **Conjugate Gradient:**

  $$d_k = -g_k + \beta d_{k-1}$$

  with β defining different CG update formulas.
- **Gradient Descent:**

  $$d = g_k$$

  with Hessian replaced with the identity matrix.

Let's think about tradeoffs at scale

The time to solve an optimization problem.

$$T(n) = N_{\text{iter}} T_{\text{iter}}(n).$$

What controls N_{iter}?

- Stopping criterion & tolerance: Objective value $f(p_k)$? Necessary condition $\nabla_p f(p_k)$?
- Iteration complexity of the algorithm
  - First-order methods decrease error linearly (add zeros after the decimal in $\|\nabla_p f(p_k)\|$ at a predictable rate near the solution)
  - Second-order methods decrease error quadratically (double zeros after the decimal in $\|\nabla_p f(p_k)\|$ at a predictable rate)

$$T(n) = N_{\text{iter}} T_{\text{iter}}(n)$$

What controls $T_{\text{iter}}(n)$?

- Evaluate the objective $f(p_k)$:
  - Depends on f (conditioning) and its implementation, but always requires information from all (distributed) parts of the objective: complexity usually no better than `allreduce()`.
- (If you can approximate $f(p_k)$ quickly and consistently without evaluating all of it, you should be looking at stochastic methods instead.)
- Evaluate the gradient $g_k = \nabla_p f(p_k)$:
  - Depends on the *adjoint efficiency* of f and its implementation.
- Solve $d = -H_k^{-1} r_k$:
  - Dependendent on *numerical conditioning* of H_k, the presence/absence of a good preconditioner, iterative linear solver efficiency, and the implementation of all of those parts.

Sensitivity Analysis

In order to use SQP algorithms, the applications must, at minimum, provide first-order derivative information.

In the broadest sense, there are two generalized ways to compute gradients.

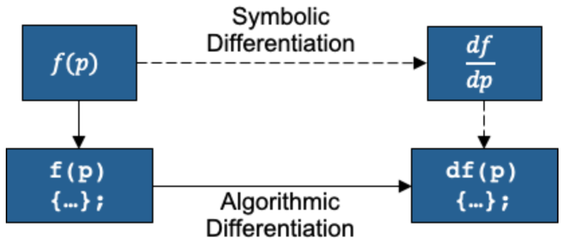
Numerical Differentiation approximates the derivative of a function using numerical methods like the *forward difference* approximation.

$$\frac{df}{dp_i} = \frac{f(p + he_i) + f(p)}{h} + \mathcal{O}(h) \quad \forall\ i = 1, 2, \ldots, N,$$

where h is the finite perturbation, e_i is the standard basis vector for the i^{th} coordinate, and $\mathcal{O}(h)$ is truncation error for the approximation.

Easy to user, requires on f itself, but computing each element of the gradient requires a separate function evaluation, (cost scales up rapidly with increasing problem sizes).

Analytical Differentiation computes the exact derivative by generating a stand-alone mathematical or algorithmic expression for the gradient.



Our example today is simple enough to do either of these by hand, but real problems generally are not.

An incomplete list of AD tools that may be useful to your application:

- ADIC (ANSI C) -- Argonne National Laboratory
- ADIFOR (Fortran77) -- Argonne National Laboratory
- OpenAD (Fortran77/Fortran95/C/C++) -- Argonne National Laboratory
- Sacado (C/C++) -- Sandia National Laboratory
- ForwardDiff.jl (Julia)
- JAX (Python)
- TOMLAB/MAD (MATLAB)
- Enzyme (LLVM IR)

Using TAO

Toolkit for Advanced Optimization (TAO) is a package of optimization algorithms and tools developed at Argonne National Laboratory and distributed with the Portable Extensible Toolkit for Scientific Computing (PETSc) library. TAO is primarily intended for **continuous gradient-based optimization** and supports PDE-constrained problems using the reduced-space formulation.

Breaking down a TAO example

```c
#include "petsc.h"

int main() {
  Tao tao;
  Vec x;

  // ... construct X (more later)
  TaoCreate(PETSC_COMM_WORLD, &tao);
  TaoSetType(tao, TAOBQNLS); // bounded, Quasi-Newton, line-search
  TaoSetSolution(tao, x);
  // ... describe f(x), g(x), H(X) (more later)
  TaoSetFromOptions(tao);
  TaoSolve(tao);
  TaoDestroy(&tao);
  // ... finalize
  return 0;
}
```

Use the search at <petsc.org> for API references.


Describing $f(x)$ by callback

```c
typedef struct { /* ... */ } AppCtx;

PetscErrorCode FormFunction(Tao tao, Vec x, PetscReal *fcn, void *ptr)
{
  AppCtx *user = (AppCtx*)ptr;
  PetscReal f_x;
  // ... user code to compute f_x
  *fcn = f_x;
  return PETSC_SUCCESS;
}

int main() {
    AppCtx user;
    // ... setup ctx
    TaoSetObjective(tao, FormFunction,  &user);
    // ...
}
```


Describing $f(x)$ by callback: performance portability

```
PetscErrorCode FormFunction(Tao tao, Vec x, PetscReal *fcn, void *ptr)
{
  AppCtx            *user = (AppCtx*)ptr;
  PetscRea      l    f_x;
  const PetscScalar *x_ptr;
  PetscMemType       memtype;

  VecGetArrayReadAndMemType(x, &x_ptr, &memtype);
  switch(memtype) {
  case PETSC_MEMTYPE_HOST:   FormFunction_Host(x_ptr, fcn, user); break;
  case PETSC_MEMTYPE_DEVICE: FormFunction_Device(x_ptr, fcn, user); break;
  // more specific cases are possible: PETSC_MEMTYPE_{CUDA,KOKKOS,SYCL}
  }
  VecRestoreArrayReadAndMemType(x, &x_ptr, &memtype);

  return PETSC_SUCCESS;
}
```


Describing $f(x)$: petsc4py bindings

```python
# PETSc src/binding/petsc4py/demo/legacy/taosolve/rosenbrock.py
class AppCtx(object):
    def __init__(self, alpha=99.0):
        self.alpha = float(alpha)

    def formObjective(self, tao, x):
        alpha = self.alpha
        ff = ... # function of x and alpha
        return ff

    # ... other callbacks

user = AppCtx()
tao = PETSc.TAO().create(PETSc.COMM_WORLD)
# ...
tao.setObjective(user.formObjective)
```


Other callbacks

```
PetscErrorCode FormObjective(Tao tao, Vec x, PetscReal *fcn, void *ptr);
PetscErrorCode FormGradient(Tao tao, Vec x, Vec g, void *ptr);
PetscErrorCode FormObjectiveAndGradient(Tao tao, Vec x, PetscReal *fcn, Vec g, void *p
tr);
PetscErrorCode FormHessian(Tao tao, Vec x, Mat H, Mat Hpre, void *ptr);
```

- `TaoSetObjectiveAndGradient()`: Many functions compute $f(x)$ and $\nabla_x f(x)$ more efficiently together than separately (particularly true of many AD systems).
- `TaoSetHessian()`: Can separately specify from a true Hessian H (correct matrix-vector products) and preconditioner H_{pre} (quickly computed sparse approximation).


`TaoSetType()` and `TaoType`

TAO implements several bound-constrained algorithm types that also solve unconstrained problems when there are no bounds defined in the problem setup. Algorithm types can be changed either via the `TaoSetType()` interface using the solver names given in the first column below, or changed at runtume with the option flag `-tao_type <solver>` using the string arguments given in the second column.

Solver Type	Option Flag	Description
`TAONLS`	`bnls`	Newton Line Search
`TAONTR`	`bntr`	Newton Trust Region
`TAOCG`	`bncg`	Nonlinear Conjugate Gradient
`TAOBNLS`	`bnls`	Bound-constrained Newton Line Search
`TAOBNTR`	`bntr`	Bound-constrained Newton Trust Region
`TAOBQNLS`	`bqnls`	Bound-constrained Quasi-Newton Line Search
`TAOBNCG`	`bncg`	Bound-constrained Nonlinear Conjugate Gradient

More useful command line arguments

Option Flag	Code API	Description
`-tao_monitor`	`TaoSetMonitor()` with `TaoMonitorDefault()`	Enable the iteration monitor for the solution
`-tao_view`	`TaoView()`	Display useful information about the solution after completion
`-tao_max_it <integer>`	`TaoSetMaximumIterations()`	Change the maximum iteration limit
`-tao_max_funcs <integer>`	`TaoSetMaximumFunctionEvaluations()`	Change the maximum number of function evaluations
`-tao_gatol <float>`	`TaoSetTolerances()`	Change the absolute convergence tolerance


Constrained optimization problems: bound constraints

$$\begin{aligned} \underset{p}{\text{minimize}} \quad & f(p) \\ \text{subject to} \quad & p_l \leq p \leq p_u \end{aligned}$$

```
/* Duplicate from solution vector and set bounds */
VecDuplicate(X, &XL);
VecSet(XL, PETSC_NINFINITY);
VecDuplicate(X, &XU);
VecSet(XU, 0.0);
TaoSetVariableBounds(tao, XL, XU);
```

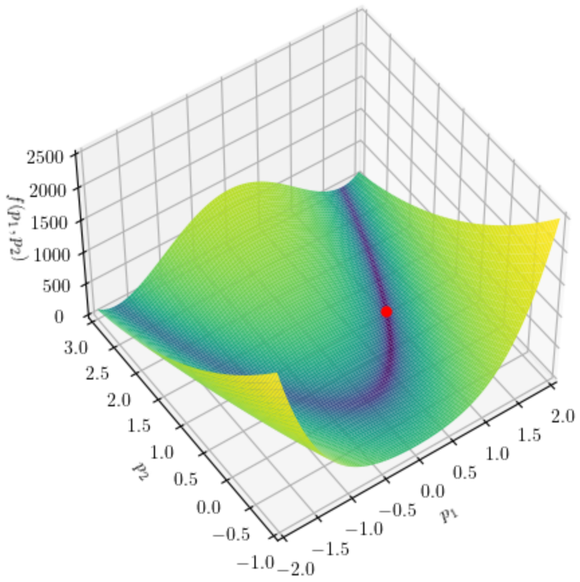

General constraints: additional callbacks

$$\begin{aligned} \underset{p}{\text{minimize}} \quad & f(p) \\ \text{subject to} \quad & c_e(p) = 0 \\ & c_i(p) \leq 0 \end{aligned}$$

```
PetscErrorCode FormEqualityConstraints(Tao tao, Vec P, Vec CE, void* ptr);
PetscErrorCode FormEqualityJacobian(Tao tao, Vec P, Mat AE, Mat AEpre, void* ptr);
PetscErrorCode FormInequalityConstraints(Tao tao, Vec P, Vec CI, void* ptr);
PetscErrorCode FormInequalityJacobian(Tao tao, Vec P, Mat AI, Mat AIpre, void* ptr);
```

Solver for general constrained problems: `TAOALMM` (Augmented Lagrangian Method of Multipliers)

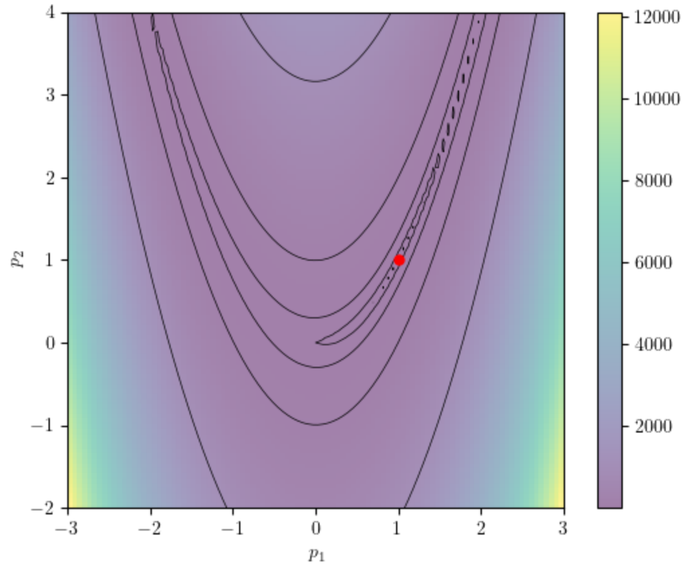
Example Problem: Multidimensional Rosenbrock



The Rosenbrock or banana function is a canonical nonconvex test problem created by Howard H. Rosenbrock in 1960 and used extensively to evaluate the performance of optimization algorithms. The original function is defined as

$$f(p_1, p_2) = (1 - p_1)^2 + 100(p_2 - p_1^2)^2,$$

with a global minimum at $f(1, 1) = 0$.



A multidimensional generalization of this problem is given by

$$f(p) = f(p_1, p_2, \ldots, p_N) = \sum_{i=1}^{N-1} \left[(1 - p_i)^2 + 100(p_{i+1} - p_i^2)^2 \right],$$

with a global minimum at $p_i = 1 \ \forall i = 1, 2, \ldots, N$.

The Hessian is a tridiagonal sparse matrix.

The hands-on example implements the multidimensional Rosenbrock with an analytical gradient and Hessian. However, TAO also provides `TaoDefaultComputeGradient()` and `TaoDefaultComputeHessian()` callbacks that utilize finite-differencing to generate the required sensitivities.

Diversion: work out $g(p)$ and $H(p)$ analytically.

Aside: verifying gradients and Hessians

From the command line, add numerical tests of gradients and Hessians to `TaoSolve()`:

```
-tao_test_gradient # test gradient numericy, reporting only errors
-tao_test_gradient_view # show inner workings of gradient test
-tao_test_hessian
-tao_test_hessian_view
```


Run the example

```
$ make multidim_rosenbrock
$ mpiexec -n 1 ./multidim_rosenbrock -tao_monitor
0 TAO,  Function value: 810081.,  Residual: 360468.
1 TAO,  Function value: 40609.1,  Residual: 44602.6
2 TAO,  Function value: 18292.1,  Residual: 26530.2
# ...
45 TAO,  Function value: 8.88666e-12,  Residual: 0.000127435
46 TAO,  Function value: 5.40212e-17,  Residual: 1.7001e-07
47 TAO,  Function value: 2.17767e-22,  Residual: 1.98927e-10
```

In this output, the `Residual` indicates the L2-norm of the gradient, $\|g_k\|_2$, at every iteration. The problem file is configured to terminate the solution when this gradient norm drops below the absolute tolerance of 10^{-5}.

More useful command line arguments (redux)

Option Flag	Code API	Description
`-tao_monitor`	`TaoSetMonitor()` with `TaoMonitorDefault()`	Enable the iteration monitor for the solution
`-tao_view`	`TaoView()`	Display useful information about the solution after completion
`-tao_max_it <integer>`	`TaoSetMaximumIterations()`	Change the maximum iteration limit
`-tao_max_funcs <integer>`	`TaoSetMaximumFunctionEvaluations()`	Change the maximum number of function evaluations
`-tao_gatol <float>`	`TaoSetTolerances()`	Change the absolute convergence tolerance
`-tao_test_gradient`	`TaoTestGradient()`	Validate the analytical gradient with finite differences at every iteration
`-tao_test_hessian`	`TaoTestHessian()`	Validate the analytical Hessian with finite differences at every iteration


Modifying the example from the command line

Option Flag	Description
`-n <integer>`	Change the problem size (default: 2)
`-fd`	Use finite-difference gradients instead of analytical
`-bound`	Activate bound constraints -- 2-D only (default: $p_1 \leq 0, \;\; p_2 \geq 0$)
`-eq`	Activate equality constraints -- 2-D only (default: $(p_1 - 1)^2 + p_2 - 3 = 0$)


Hands-on Activities

1. Change the TAO algorithm to nonlinear conjugate gradient method using `-tao_type bncg` and to truncated Newton using `-tao_type bnls`. Compare convergence against the default quasi-Newton method (`-tao_type bqnls`).


2. Increase the problem size with the `-n <size>` argument (default size is 2) and evaluate its impact on convergence.

    - Repeat Activity 2 with different TAO algorithms. Do they all exhibit the same scaling?

3. Solve the problem with the finite difference gradient using the `-fd` argument. Evaluate convergence and solution time with increasing problem size.

4. Try running the problem in parallel with `mpiexec -n <# of processes> ./multidim_rosenbrock ...`. Why does running in parallel slow the solution down at small problem sizes? How large should the problem be to observe a speedup in parallel runs?

    - Repeat Activity 4 with different TAO algorithms. Are the break-even points in size vs. performance the same?


5. Run the problem with `-bound` flag to enable $p_1 \leq 0$ and $p_2 \geq 0$ constraints.

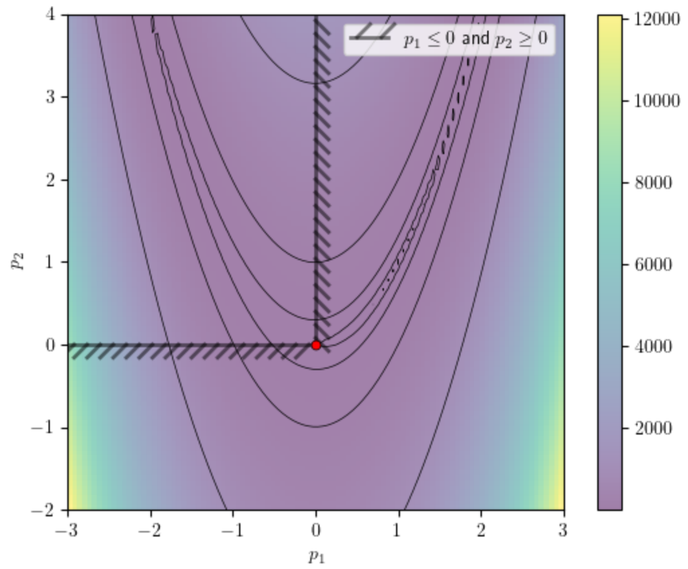
- Change the starting point and evaluate how it affects the convergence. Is there a difference between starting from the feasible versus non-feasible space?
- Repeat Activity 5 with different bound-constrained TAO algorithms.

6. Run the problem with `-eq` flag to enable the $(p_1 - 1)^2 + p_2 = 3$ constraint.

   - Change the starting point to $(-1, -1)$ and evaluate whether you recover the same solution as before. If not, why?
   - Combine equality constraints with bound constraints and try changing the starting point back to $(10, 10)$. Which solution did you converge to?

7. ADVANCED: Change the constraint definition. Come up with constraints that are valid for the multidimensional Rosenbrock problem.

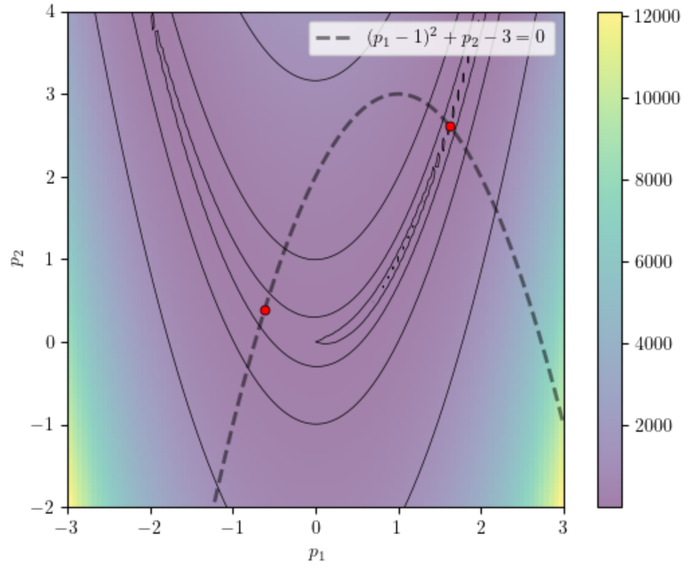
Notes on bound constraints



Introducing the bound constraints $p_1 \leq 0$ and $p_2 \geq 0$ forces the solution to a new global minimum at $f(0, 0) = 1.0$. This minimum is a trivial solution where the bounds for both optimization variables. are active.

Notes on equality constraints

The quadratic equality constraint $(p_1 - 1)^2 + p_2 - 3 = 0$ presents two local minima instead of the original global minimum. These two minima lie at $f(1.62, 2.62) = 0.38$ and $f(-0.62, 0.38) = 2.62$. On multi-modal problems such as this, gradient-based optimization methods converge to the local minimum closest to the starting point. In this hands-on example, we start our solution with an initial guess of $(10, 10)$ and converge to $f(1.62, 2.62) = 0.38$. A solution that starts at $(-1, -1)$ converges to $f(-0.62, 0.38) = 2.62$ instead.



Combining the equality and bound constraints eliminates one of the two local minima and forces the solution to always converge to the now-global minimum at $f(-0.62, 0.38) = 2.62$ regardless of the starting point.